

METHOD FOR STREAMLINING ERROR CORRECTION CODE COMPUTATION
WHILE READING OR PROGRAMMING
A NAND FLASH MEMORY

PRIORITY CLAIM

[0001] This application claims priority to EPO Patent Application Serial No. 03293051.3, filed on December 4, 2003, entitled "STREAMLINING ECC COMPUTATION WHILE READING OR PROGRAMMING A FLASH MEMORY," incorporated herein by reference.

BACKGROUND

[0002] Computers may comprise various types of memory for storing data. Each of these types of memory has various advantages that suit different applications. Flash memory, for example, is a solid state device used for fast and efficient memory storage. Examples of flash memory comprise a computer's BIOS chip, a memory stick, compact flash cards, SmartMedia™ cards and PCMCIA memory cards.

[0003] High-speed programming and erasing capabilities, low cost, ease of memory expansion, long lifespan and a file memory architecture make NAND Flash memory particularly useful in data-centric applications. NAND Flash memory has an input-output ("I/O") interface and uses a protocol that includes fast read/write/erase commands, addresses and data. Despite the advantages, NAND Flash memory does not permit easy access to a random memory address. NAND Flash memory also may be prone to low reliability due to random errors generated by physical effects in the geometry of the NAND gates in the memory. Such random errors also may be caused by an excessive number of read, write and erase cycles.

[0004] To correct such errors, a host processor may use a NAND Flash controller ("controller") to generate an error correction code ("ECC"). For the purposes of error checking and correction, an ECC may be generated when programming a NAND Flash

with a particular data. Another ECC may be generated when reading the particular data from the NAND Flash. The two ECCs subsequently may be compared to locate and correct any differences which may represent errors caused by the NAND Flash memory.

[0005] Because NAND Flash memory typically is accessed in blocks of data, the controller may be forced to wait until a full block of data has been accessed before computing the ECC, thereby incurring a penalty in performance. Additionally, because a controller may have a limited number of registers in which to store completed ECC computations, the controller may pause several times to dump the contents of the register(s) to clear register space for new, completed ECC computations, resulting in further performance penalties. A method to compute and store the ECC without suffering a loss in performance is desirable.

BRIEF SUMMARY

[0006] The problems noted above are solved in large part by a method for streamlining ECC computation while reading or programming a NAND Flash memory. One exemplary embodiment may comprise transferring a data block between a flash memory and a memory controller and computing an error correction code while transferring the data block.

BRIEF DESCRIPTION OF THE DRAWINGS

[0007] For a detailed description of exemplary embodiments of the invention, reference will now be made to the accompanying drawings in which:

[0008] Figure 1a illustrates a block diagram of a NAND Flash memory controller, a NAND Flash memory and a host processor in accordance with certain embodiments of the invention;

[0009] Figure 1b illustrates a block diagram of a register switching mechanism in accordance with certain embodiments of the invention;

[0010] Figure 2 illustrates a second block diagram of a NAND Flash memory controller, a NAND Flash memory and a host processor in accordance with certain embodiments of the invention;

[0011] Figure 3 illustrates a block diagram of ECC register memory allocation in accordance with certain embodiments of the invention;

[0012] Figure 4a illustrates a flow diagram of a NAND Flash memory programming operation in accordance with certain embodiments of the invention; and

[0013] Figure 4b illustrates a flow diagram of a NAND Flash memory read operation in accordance with certain embodiments of the invention.

NOTATION AND NOMENCLATURE

[0014] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, various companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to... ." Also, the term "couple" or "couples" is intended to mean either an indirect or direct electrical connection. Thus, if a first device couples to a second device, that connection may be through a direct electrical connection, or through an indirect electrical connection via other devices and connections.

DETAILED DESCRIPTION

[0015] The following discussion is directed to various embodiments of the invention. Although one or more of these embodiments may be preferred, the embodiments disclosed should not be interpreted, or otherwise used, as limiting the scope of the disclosure, including the claims. In addition, one skilled in the art will understand that the following description has broad application, and the discussion of any embodiment is meant only to be exemplary of that embodiment, and not intended to intimate that the scope of the disclosure, including the claims, is limited to that embodiment.

[0016] Figure 1a illustrates a NAND Flash memory controller 100 coupled to a host processor 135 and a NAND Flash memory 114. The NAND Flash memory controller 100 may transfer data to and receive data from the NAND Flash memory 114. In some situations, data stored in the NAND Flash memory 114 may become corrupted.

[0017] The NAND Flash memory controller 100 may monitor data for accuracy by computing the ECC for each data transaction involving the NAND Flash memory 114. Specifically, the NAND Flash memory controller 100 may calculate the ECC upon writing a particular data into the NAND Flash memory 114. The ECC may be stored with the

data. On a read transaction, the NAND Flash memory controller 100 may retrieve the target data and previously computed ECC, recalculate the ECC based on the retrieved data, and subsequently the host processor compares the newly recalculated ECC with the previously calculated ECC to locate and correct errors or discrepancies in the data. Because, in accordance with conventional NAND Flash memory, data entered into or retrieved from the NAND Flash memory 114 is in blocks of a finite size, the ECC may not be computed until the entire block has been written or read. Thus, the host processor 135 may suffer a loss in performance. The block size may be 256 or 512 bytes, but other block sizes are within the scope of this disclosure.

[0018] A full page of NAND Flash memory 114 may comprise a plurality of blocks of data. Thus, the NAND Flash memory controller 100 may read or write a plurality of blocks when reading or writing a full page of data. After each block of is read or written, a typical NAND Flash memory controller 100 may calculate the ECC, transfer the ECC to the host processor 135, and then dump the ECC information. The amount of time lost in completing such processes may cause the host processor 135 to suffer a further loss in performance, especially if the transfer between the host processor and the controller is performed by a direct memory access device ("DMA"). In accordance with the preferred embodiment of the invention as described below, serialization of the ECC computation process and the addition of an automatic ECC register switching mechanism compensates for such performance losses.

[0019] One algorithm commonly used to compute an ECC is the Hamming algorithm. While the Hamming algorithm may be used with any number or bits and any amount of data, the following example illustrating the Hamming algorithm uses four bits and four packets of data.

[0020] The Hamming algorithm computes a series of parity bits using the XOR logic function (hereafter denoted by " \wedge "). Table 1 below illustrates data rows A, B, C and D.

Table 1. XOR operations performed to compute parity bits.

A	A3	A2	A1	A0	X
B	B3	B2	B1	B0	Y
C	C3	C2	C1	C0	Z
D	D3	D2	D1	D0	T
R	R3	R2	R1	R0	

[0021] Each column of row R contains the result of the XOR operation performed on all data present in the column. For example, $R3 = A3 \wedge B3 \wedge C3 \wedge D3$. The X, Y, Z and T values are the results of the XOR operation performed on all bits present in the associated row. For example, $X = A3 \wedge A2 \wedge A1 \wedge A0$. Table 2 below includes all the XOR values for Table 1.

Table 2. XOR computations performed for rows and columns of Table 1.

$R0 = A0 \wedge B0 \wedge C0 \wedge D0$ $R1 = A1 \wedge B1 \wedge C1 \wedge D1$ $R2 = A2 \wedge B2 \wedge C2 \wedge D2$ $R3 = A3 \wedge B3 \wedge C3 \wedge D3$	$X = A3 \wedge A2 \wedge A1 \wedge A0$ $Y = B3 \wedge B2 \wedge B1 \wedge B0$ $Z = C3 \wedge C2 \wedge C1 \wedge C0$ $T = D3 \wedge D2 \wedge D1 \wedge D0$
--	--

[0022] The series of parity bits generated by the Hamming algorithm may be used to represent the ECC. Tables 3a and 3b illustrate the generation of parity bits using the values of Table 2. Specifically, Table 3a illustrates the generation of column parities and Table 3b illustrates the generation of line parities.

Table 3a. Column parity bits generated using the results of Table 2.

$P1o = R3 \wedge R1$ $P1e = R2 \wedge R0$	$P2o = R3 \wedge R2$ $P2e = R1 \wedge R0$
--	--

Table 3b. Line parity bits generated using the results of Table 2.

$P4e = X \wedge Z$	$P4o = Y \wedge T$	$P8e = X \wedge Y$	$P8o = Z \wedge T$
--------------------	--------------------	--------------------	--------------------

[0023] The series of parity bits used to represent the ECC may be defined as:

$$P8oP4oP2oP1oP8eP4eP2eP1e, \quad (1)$$

where each parity bit of expression (1) is placed next to another parity bit of expression (1) to form an 8-bit parity that protects 16 bits of data. Parities may be of any appropriate size to protect a particular amount of data. Parity bits comprising "e" represent even data bits and parity bits comprising "o" represent odd data bits.

[0024] Because the Hamming algorithm used to compute the ECC uses XOR gates, the algorithm may be refined so that the ECC is computed while data is simultaneously read or written to the NAND Flash memory 114. Such a refined algorithm may be referred to as a serialized algorithm. The serialized algorithm may be illustrated in context of the computation of a parity bit. While the process illustrated in the following example is shown in the context of a read operation, the same process also may be applied to a write operation. As shown in Table 3a:

$$P1o = R3 \wedge R1 = (A3 \wedge B3 \wedge C3 \wedge D3) \wedge (A1 \wedge B1 \wedge C1 \wedge D1). \quad (2)$$

Because the XOR function is mathematically commutative and associative, expression (2) also may be expressed as:

$$P1o = (((A3 \wedge A1) \wedge B3 \wedge B1) \wedge C3 \wedge C1) \wedge D3 \wedge D1. \quad (3)$$

The serialized algorithm comprises computing the parity bit while data is being read. Continuing with this example, if data row A of Table 1 is the first to be read from flash memory, then

$$A3 \wedge A1 \quad (4)$$

may be computed and stored in any suitable storage area (e.g., a register or RAM). If data row B is the next to be read, then

$$B3 \wedge B1 \quad (5)$$

may be computed. The modified algorithm is of a serialized nature and thus comprises computing as much of expression (3) as possible. Specifically, the XOR function may be applied to expressions (4) and (5) to produce

$$(A3^A1) \wedge (B3^B1) \quad (6)$$

and the result may be stored in any suitable storage area. If data row C is next, then

$$C3^C1 \quad (7)$$

may be computed. The XOR function subsequently may be applied to expressions (6) and (7) to produce

$$((A3^A1) \wedge B3^B1) \wedge C3^C1 \quad (8)$$

which may be stored in some temporary storage area. If data row D is the last to be read, then

$$D3^D1 \quad (9)$$

may be computed. The XOR function subsequently may be applied to expressions (8) and (9) to produce

$$(((A3^A1) \wedge B3^B1) \wedge C3^C1) \wedge D3^D1 \quad (10)$$

which is the same as expression (3). The same process may be performed for all column parities.

[0025] A similar process may be used to compute the line parities of Table 3b. By way of an example, for a new ECC computation,

$$\begin{aligned} P4e(0) &= 0 \\ P4o(0) &= 0 \\ P8e(0) &= 0 \\ P8o(0) &= 0. \end{aligned} \tag{11}$$

If a new data row A is read,

$$X = A3^A2^A1^A0 \tag{12}$$

may be computed. The XOR function then may be applied to groups (11) and (12) to produce

$$\begin{aligned} P4e(1) &= P4e(0) \wedge X = 0 \wedge X = X \\ P4o(1) &= P4o(0) \wedge 0 = 0 \wedge 0 = 0 \\ P8e(1) &= P8e(0) \wedge X = 0 \wedge X = X \\ P8o(1) &= P8o(0) \wedge 0 = 0 \wedge 0 = 0. \end{aligned} \tag{13}$$

Once data row B is read,

$$Y = B3^B2^B1^B0 \tag{14}$$

may be computed. The XOR function then may be applied to groups (13) and (14) to produce

$$\begin{aligned} P4e(2) &= P4e(1) \wedge 0 = X \wedge 0 = X \\ P4o(2) &= P4o(1) \wedge Y = 0 \wedge Y = Y \\ P8e(2) &= P8e(1) \wedge Y = X \wedge Y = X \wedge Y \\ P8o(2) &= P8o(1) \wedge 0 = 0 \wedge 0 = 0. \end{aligned} \tag{15}$$

Assuming data row C is next,

$$Z = C3^C2^C1^C0 \tag{16}$$

may be generated. The XOR function then may be applied to groups (15) and (16) to produce

$$\begin{aligned}
 P4e(3) &= P4e(2) \wedge Z = X \wedge Z &= X \wedge Z \\
 P4o(3) &= P4o(2) \wedge 0 = Y \wedge 0 &= Y \\
 P8e(3) &= P8e(2) \wedge 0 = X \wedge Y \wedge 0 = X \wedge Y \\
 P8o(3) &= P8o(2) \wedge Z = 0 \wedge Z &= Z.
 \end{aligned} \tag{17}$$

If data row D is read last,

$$T = D3 \wedge D2 \wedge D1 \wedge D0 \tag{18}$$

may be computed. The XOR function then may be applied to groups (17) and (18) to produce

$$\begin{aligned}
 P4e(4) &= P4e(3) \wedge 0 = X \wedge Z \wedge 0 = X \wedge Z \\
 P4o(4) &= P4o(3) \wedge T = Y \wedge T &= Y \wedge T \\
 P8e(4) &= P8e(3) \wedge 0 = X \wedge Y \wedge 0 = X \wedge Y \\
 P8o(4) &= P8o(3) \wedge T = Z \wedge T &= Z \wedge T.
 \end{aligned} \tag{19}$$

In this manner, the serialized algorithm computes the parity bits representative of the ECC while data is read, whereas the original Hamming algorithm computes the parity bits after all data has been read. Thus, while both algorithms may result in a series of parity bits representative of the ECC as illustrated in expression (1), the serialized algorithm is more efficient than the original Hamming algorithm. One embodiment may comprise transferring a data block between a flash memory 114 and a memory controller 100 and computing an ECC for the data block while transferring the data block. Whereas the ECC of a data block is commonly computed after transferring the data block, this embodiment comprises computing the ECC of a data block while simultaneously transferring the data block, thereby increasing efficiency and significantly reducing performance loss.

[0026] In addition to the serialized algorithm, performance speed also may be increased by using an automatic ECC register switching mechanism. In many existing systems, the ECC is computed and then stored in a single ECC register. Such a register typically has a capacity of protecting 256 bytes or 512 bytes of memory. Thus, each time the register becomes full of data, the data must be saved to some external memory and the contents of the register must be dumped to create space for new ECC computations. This process

is time consuming and inefficient, since a register may be dumped several times over the duration of a read or write operation.

[0027] The preferred embodiment of the invention comprises an automatic ECC register switching mechanism. As shown in Figure 1b, the automatic ECC register switching mechanism 190 may comprise, among other things, a plurality of ECC registers 218-234 and a switch 238 to select one of the ECC registers 218-234, such as ECC register 218. Once an ECC register 218 becomes full with blocks of data, the switch 238 may select a second, empty ECC register 220 to store incoming ECC computations. Once the second ECC register 220 is full of data, the switch 238 may select a third, empty ECC register 222 to store incoming ECC computations. The process may continue in this fashion until all available ECC registers 218-234 are full of data, at which point the data may be saved to some appropriate, external memory and the ECC registers 218-234 may be dumped. The automatic switching mechanism 190 is more efficient than commonly-used technology because of the ability to store large amounts of data without pausing to empty the ECC registers 218-234.

[0028] The serialization of the ECC computation process and the automatic ECC register switching mechanism 190, as described above, may be combined to increase efficiency and avoid penalties in performance when computing the ECC. Figure 2 illustrates a NAND Flash controller 200 coupled to a host processor 236 and a NAND Flash memory 214 by way of an interface bus 212 and an I/O bus 216, respectively. The NAND Flash controller 200 may comprise an ECC engine 202, a memory interface 204, a sequencer 206, a FIFO 208, a processor interface 210, ECC registers 218-234 and an ECC register switch 238.

[0029] The processor interface 210 is the primary interface for the host processor 236 to communicate with the controller 200. The processor interface 210 comprises various registers to read or program the NAND Flash memory 214. The FIFO 208 may comprise a dedicated bank of registers that acts as a buffer to read or program the NAND Flash memory 214. The sequencer 206 may be a state machine to send data, addresses and commands to the NAND Flash memory 214. The memory interface 204 controls the timing of the signals that couple to the NAND Flash memory 214. The ECC engine 202 preferably computes the ECC for read and write operations.

[0030] The ECC engine 202 or other appropriate entity may set the switch 238 to select ECC register 218. When the host processor 236 reads the NAND Flash memory 214, a copy of the data may be routed to the ECC engine 202. The ECC engine 202 subsequently begins computing the ECC for the data using the serialized algorithm as described above. The ECC engine 202 preferably stores the ECC computations in ECC register 218. The ECC engine 202 may continue to compute the ECC for new data and store the results in ECC register 218 until no further data is available. Once either 256 bytes or 512 bytes have been reached (as indicated by an ECC256/512 signal), the switch 238 may point to ECC register 220 and the ECC computation process may continue. ECC computations may continue to be generated and stored in the ECC register 220 until 256 bytes or 512 have been reached (as indicated by the ECC256/512 signal to the ECC engine 202), whereupon the switch 238 may be set to ECC register 222. This process may be repeated until all ECC registers 218-234 are full of data and must be emptied. Because the size of a full page of NAND Flash memory is variable, the number of ECC registers present in the controller 200 also may vary.

[0031] When a full page of the NAND Flash memory 214 has been read, the ECC computations may be ready in the ECC registers 218-234. The host processor 236 may read the ECC registers 218-234 and the ECC registers 218-234 subsequently may be emptied.

[0032] Referring now to Figures 2 and 3, Figure 3 provides an illustrative embodiment wherein a full page of NAND Flash memory 214 is defined as 2KB of data 318 plus a "spare area" of 64 bytes 316 and the ECC256/512 signal to the ECC engine 202 is selected to be 256 bytes. An initial 256 bytes of data 300 may be received by the ECC engine 202. The ECC engine 202 may compute the ECC for the initial 256 bytes of data 300 and store the results in ECC register 218. The ECC engine 202 then may receive a second 256 bytes of data 302, generate the corresponding ECC computations and store the ECC computations in ECC register 220. The process may continue in this fashion until the ECC is computed for a last 64 bytes of data 316 and stored in ECC register 234. In this manner, a full page of NAND Flash memory 214 may be calculated without any delays for calculating the ECC. This process may accommodate any data size.

[0033] Figure 4a provides a flowchart of an illustrative NAND Flash memory 214 programming operation. The process may begin with the selection of either 256 bytes or 512 bytes as the ECC computation size (block 402). The ECC computation size may be selected by an end user, a processor, or any other suitable entity. The NAND Flash controller 200 then may initiate the programming process by sending a “program” command to the NAND Flash memory 214 (block 404). The starting address of the NAND Flash memory 214 where data may be written also may be sent to the NAND Flash memory 214 (block 406). Data then may be sent to the NAND Flash memory 214 and the ECC Engine 202 may concurrently compute the ECC and store the ECC in the current ECC register 218-234 (block 408). If 256 bytes or 512 bytes of data (as selected in block 402) have been stored in the current ECC register 218-234 (block 410), then switch 238 may select the next ECC register 218-234 as the current register (block 412). If the end of a full page of NAND Flash memory 214 has been reached (block 414), the ECC registers 218-234 may be read (block 416) and saved in a spare area of NAND Flash memory 214 (block 418). If there exists additional data to be programmed to the NAND Flash memory 214 (block 420), the ECC registers may be dumped (block 424), the data may be sent to the NAND Flash memory and the ECC may be concurrently computed and stored in an ECC register 218-234 (block 408). The process may continue until all data intended for transmission are sent to the NAND Flash memory 214. Otherwise, the programming process may be complete (block 422).

[0034] Figure 4b provides a flowchart of an illustrative NAND Flash memory 214 read operation. The process may begin with the selection of either 256 bytes or 512 bytes as the ECC computation size (block 452). The ECC computation size may be selected by an end user, a processor, or any other suitable entity. The NAND Flash controller 200 may initiate the reading process by sending a “read” command to the NAND Flash memory 214 (block 454). The starting address of the NAND Flash memory 214 from which data may be read also may be sent to the NAND Flash memory 214 (block 456). Data then may be read from the NAND Flash memory 214 and the ECC engine 202 may concurrently compute the ECC and store the ECC in the current ECC register 218-234 (block 458). If 256 bytes or 512 bytes of data (as selected in block 452) have been stored in the current ECC register 218-234 (block 460), then switch 238 may select the next

ECC register 218-234 as the current register (block 462). If the end of a full page of NAND Flash memory 214 has been reached (block 464), the ECC registers 218-234 may be read (block 466) and compared with the ECC registers 218-234 stored in the spare area of NAND Flash memory 214 as indicated in block 418 of Figure 4A (block 468). Such a comparison between the ECC generated when a particular data was written and the ECC generated when the same data was read enables the host processor 135 to determine whether any data errors were precipitated by the NAND Flash memory 214. The host processor 135 may correct existing data errors using any of a variety of external software.

[0035] If there exists additional data to be read from the NAND Flash memory 214 (block 470), the ECC registers may be cleared (block 474), the data may be read from the NAND Flash memory 214 and the ECC may be concurrently computed and stored in an ECC register 218-234 (block 458). The process may continue until all data intended for transmission are read from the NAND Flash memory 214. Otherwise, the reading process may be complete (block 472).

[0036] The above discussion is meant to be illustrative of the principles and various embodiments of the present invention. Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.